Preprint
UCRL-JC-150927

Co-ordinated Coscheduling in Clusters through a Generic Framework

S. Agarwal, G. S. Choi, C. R. Das, A. B. Yoo, and S. Nagar

This article was submitted to
Association for Computing Machinery SIGMETRICS Conference,
Marina Del Rey, California, June 15-19, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

November 4, 2002

Co-ordinated Coscheduling in Clusters through a Generic Framework. *

Saurabh Agarwal, Gyu Sang Choi, Chita R. Das
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{sagarwal,gchoi,das}@cse.psu.edu

Andy B. Yoo
Lawrence Livermore National Laboratory
Livermore, CA 94551
yoo2@llnl.gov

Shailabh Nagar
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
nagar@us.ibm.com

Abstract

Communication-driven scheduling is known to be an effective technique to improve the performance of parallel workloads in time-sharing clusters. Although several such coscheduling algorithms have been proposed, to our knowledge, none of these techniques have been adopted in commercial systems. We believe this is primarily because many of these algorithms has not been exhaustively tested on real systems in presence of mixed workloads, and hence, have not been demonstrated as a favorable alternative to the traditional, batch scheduling. Moreover, practical issues like lack of a methodological approach to efficiently implement, port or reuse the necessary software have dissuaded designers from including coscheduling as a feature in the mainstream system software layer.

In this paper, we attempt to fill these crucial voids by addressing several key issues. First, we propose a generic framework for deploying coscheduling techniques by providing a reusable and dynamically loadable kernel module. Second, we implement three prior dynamic coscheduling algorithms (*Dynamic coscheduling* (DCS), *Spin Block* (SB) and *Periodic Boost* (PB)) and a new coscheduling technique, called *Co-ordinated coscheduling* (CC), using the above framework. Then, we demonstrate the effectiveness of these strategies by implementing a prototype on a Myrinet connected 16-node Linux cluster that uses industry standard Virtual Interface Architecture (VIA) as the user-level communication abstraction. Our indepth performance analysis using a variety of workloads reveals several interesting observations and better insights on the relative merits of the four coscheduling schemes. First, in contrast to some previous results, where PB was shown as the best performer, we observe that SB and the proposed CC scheme outperform all other techniques on a Linux platform that dominates the current market place. This leads to the second conclusion that the choice of the native scheduler plays a significant role in deciding a competitive coscheduling strategy. Third, we find that SB and CC schemes are effective alternatives to batch processing even at a reasonable multiprogramming level of 4 to 6. Finally, we show that despite being an early prototype attempt, the proposed coscheduling scheme (CC) provides equal or slightly better performance than SB with the added advantages of flexibility, generality, and potential for incorporating specialized services such as QoS.

*This research was supported in part for DOE by UC/LLNL under contract no. W-7405-Eng-48.

1 Introduction

The availability of high bandwidth, low-latency networks [1, 2, 3, 4], supported by efficient user-level communication protocols [5, 6, 7, 8, 23], has made clusters [9] an attractive and cost effective alternative to traditional multiprocessor systems. However, in the context of high performance parallel computing, advantages of fast networks and low-overhead communication protocols can be nullified if the underlying process scheduling technique is not efficient. Optimally scheduling processes of a parallel job onto various nodes of a parallel system has always been a challenging problem. However, unlike a tightly-coupled multiprocessor environment, where a global scheduler schedules jobs across all the nodes, process scheduling in a cluster environment becomes more complex and challenging, primarily due to the autonomy of individual nodes. Therefore, process scheduling has re-surfaced as a critical research focus for cluster systems.

The possible alternatives to concurrently schedule processes of a parallel job on individual nodes, called *coscheduling* [10], span from a naive local scheduling to more sophisticated techniques like gang scheduling [11, 12, 13] (sometimes also called coscheduling) or communication-driven coscheduling [14, 15, 16]. In local scheduling, processes of a parallel job are independently scheduled by the native scheduler on each node, without any effort to coschedule them. Although simpler to implement, local scheduling can be very inefficient for running parallel jobs that need process coordination. The Gang scheduling [11, 12, 13], on the other hand, uses explicit global knowledge to schedule all the processes of a job simultaneously. Gang scheduling is an ideal coscheduling scheme, whose effectiveness has been demonstrated for tightly-coupled multiprocessors through some successful deployments in commercial systems [1]. However, it may not be a scalable option for a cluster environment because, the explicit communication, required to exchange global information amongst local schedulers, is quite expensive. Therefore, a few other coscheduling alternatives such as dynamic coscheduling (DCS) [14, 17, 18], implicit coscheduling (ICS) [15], Spin Block (SB) [16, 19] and periodic boost (PB) [16, 19] have been proposed recently for clusters.

These coscheduling techniques (DCS, ICS, SB, PB) rely on the communication behavior of the applications to schedule processes of a job simultaneously. (Hence, we call them communication-driven coscheduling algorithms.) Using the efficient user-level communication protocols such as U-Net [5], Active Messages [7] and Fast Messages [6], they have been shown to be quite efficient. However, all these studies have been conducted in various restricted environments as summarized in Section 2, and to the best of our knowledge, none of these these techniques have yet made their way into commercial/large user-base platforms. Most of the commercial clusters use batch scheduling systems [20, 21]. This remains true despite the fact that batch systems suffer from poor response times in a time-sharing environment.

From the above discussion, we observe that commercial viability of these dynamic coscheduling techniques is conspicuously missing, and thus, needs careful investigation. In this context, in addition to being able to provide high performance, ease of implementation, portability, and scalability of a scheduling technique are vital for adapting it on different hardware and software platforms.

The research, presented in this paper, is thus motivated by a number of factors. First, while Linux clusters are becoming predominant platforms for various type of servers, none of the coscheduling techniques has been implemented on a Linux platform using user-level communication paradigm. We believe that demonstrating the effectiveness of coscheduling on a Linux cluster will be an important step in the coscheduling research. Second,

[1] IBM ASCI BLUE, ASCI WHITE etc

availability of a standard and modular framework that can be used to implement various coscheduling strategies and can easily be ported on a variety of platforms with minimal effort, will be a significant contribution. Third, choice of a user-level communication layer is critical for conforming to InfiniBand Architecture (IBA) [22], which may become the future interconnect standard for clusters. We investigate these algorithms on the current industry standard (for user-level communication) - the Virtual Interface Architecture [8, 23] (VIA), with the objective of porting them to IBA [2]. Next, the impact of the native OS scheduler on the coscheduling techniques needs careful examination. Moreover, a closer look at the existing techniques may reveal new directions for further research. Finally, as quality-of-service (QoS) provisioning in clusters is becoming critical to support various real-time applications, it would be useful if the coscheduling algorithms can be extended to provide end-to-end QoS support.

The paper addresses most of these concerns by introducing a generic framework that can be used to implement a coscheduling algorithm with minimal effort. The framework presents well defined interfaces to the native Linux scheduler, application level libraries like the Message Passing Interface (MPI) [40] and the Virtual Interface Provider Library (VIPL) [8], the network interface card's (NIC) device driver, and its firmware. This was possible by implementing coscheduling *policies* in a stand alone loadable kernel module and leaving it up to the designers to implement the necessary *mechanism* in the device driver/firmware through a proper interface.

The framework has been implemented on a 16-node Linux cluster connected through Myrinet. We have done significant extension of the Berkeley VIA [23] driver, firmware and the API, to run on our cluster, and we believe, this is the first work that presents experiments on a Myrinet connected 16-node VIA platform. The prior coscheduling algorithms (DCS, PB, and SB) have been implemented using this framework. Analysis of the prior policies indicated that spin-based schemes like DCS and PB waste significant CPU time, which can be effectively utilized by other processes through a blocking mechanism. Thus, we believe that asynchronous, interrupt-driven techniques like SB can boost application performance especially because the interrupt and context switch overheads in optimized kernels like Linux have reduced drastically.

In view of this, we present a novel coscheduling algorithm, called Co-ordinated coscheduling (CC), which takes into account spinning time optimizations at both the sender and receiver ends. The sender side optimization is based on the premise that as the network size and the multi programming level (MPL) increases, a process may spin unnecessarily for a long time before the data can be pushed to the wire. Hence, it may benefit to block the process if the data cannot be sent in a reasonable amount of time, and allow another process to use the CPU. The original process can be scheduled after the data is put on the wire. The receiver side blocking, on the other hand, helps in optimizing the CPU time if the data is not received during the spin time. Unlike the SB design [19], the proposed CC scheme also uses a boosting mechanism to schedule an appropriate receiver process. All these schemes have been tested exhaustively using NAS parallel benchmarks along with a mix of CPU and I/O intensive serial applications.

Our experiments reveal several significant results. First, in the presence of a fair and fast scheduler as found in Linux 2.X.X, blocking-based schemes like CC and SB perform significantly better than spin-only based schemes like PB or DCS, contrary to some previous results [16, 24, 34]. Second, the above observation leads us to the conclusion that choice of the native scheduler has a significant impact on the coscheduling policy. Third, we demonstrate that it is possible to provide a competitive or better performance than batch scheduling through

[2] Note that the communication architecture of IBA is derived from that of VIA, and thus they have a striking similarity.

coscheduling schemes like CC and SB, at a much better user response time. This is especially true above a moderate MPL of four to six. Next, we find that the presence of CPU intensive sequential jobs have lesser impact on parallel workloads than I/O intensive jobs. Moreover, we also observe that coscheduling policies have a much lesser impact on I/O jobs as compared to CPU intensive jobs. Finally, we show that SB and CC are viable coscheduling alternatives for a Linux platform and we argue that although the CC scheme exhibits similar or slightly better performance than the SB algorithm, it has the potential to perform better on larger clusters. Besides, it provides additional advantages like generality and flexibility to incorporate customized boosting mechanisms.

The rest of this paper is organized as follows. In Section 2, we summarize the prior coscheduling research. Section 3 outlines the design of our generic framework, while Section 4 discusses the proposed CC mechanism. The performance results are analyzed in Section 5, followed by the concluding remarks in the last Section.

2 Related Work

Scheduling of parallel processes onto processors of a parallel machine has been an important research topic, and a myriad of scheduling techniques have been proposed for multiprocessors [25, 26, 27, 28]. However, design of such schedulers for clusters is much more challenging, primarily because of the individual node autonomy. Understanding the practical limitations in implementing gang scheduling [11, 12, 13] and perfect coscheduling [10] for such systems, recently a few implementations [16, 15, 14] and several simulation driven studies [29, 17, 30, 31] have proposed communication-driven coscheduling techniques to efficiently schedule parallel jobs.

Dynamic coscheduling [17, 14] (DCS) optimistically assumes that if a message arrives from a node at the NIC, the sender is scheduled and hence, the NIC firmware interrupts the host to schedule the corresponding receiver process if it is not already scheduled. Implicit coscheduling [29, 15] (ICS), in contrast, after sending a message, spins for some time to wait for a reply. If the reply does not arrive within that time, the sender blocks. Although it appears that ICS is a sender side optimization, a closer look reveals that it is really a receive on which the scheme operates, with an added condition that the receive it applies to is after a previous send. In this sense, it is coarser grained as compared to DCS. Conceptually, the difference is that while DCS deals with all message arrivals, ICS deals with messages that arrive for blocked processes only. This scheme is shown to be quite efficient in a tightly-coupled programming model like Split-C/Active Messages [7], where communication events are synchronous reads or writes, which means that every send has an immediate reply associated with it. However, in a loosely-coupled programming model like MPI [32], asynchronous send-receive communication primitives are used. This releases the requirement of a reply for each send and allows a sender to progress until the data is actually needed. Implications of ICS algorithm in such a scenario have not been considered.

A good approximation of the ICS scheme, as implemented in [19], is spin block (SB), where a receiver always spins for some time to wait for a message before it blocks itself, irrespective of weather or not it did a send earlier. The spin time is judiciously selected based on the network latency and protocol processing overheads. Being more appropriate for the MPI-based asynchronous message passing programming model, we chose to only implement SB in our experiments. Contrary to DCS or ICS, Nagar et. al proposed a different approach, called Periodic Boost (PB) [16] , which is based on polling instead of interrupt. However, similar to previous schemes, it is also a receiver side optimization. In PB, a kernel entity periodically polls communication end-points on the host memory to check if there are any pending messages, and boosts the corresponding

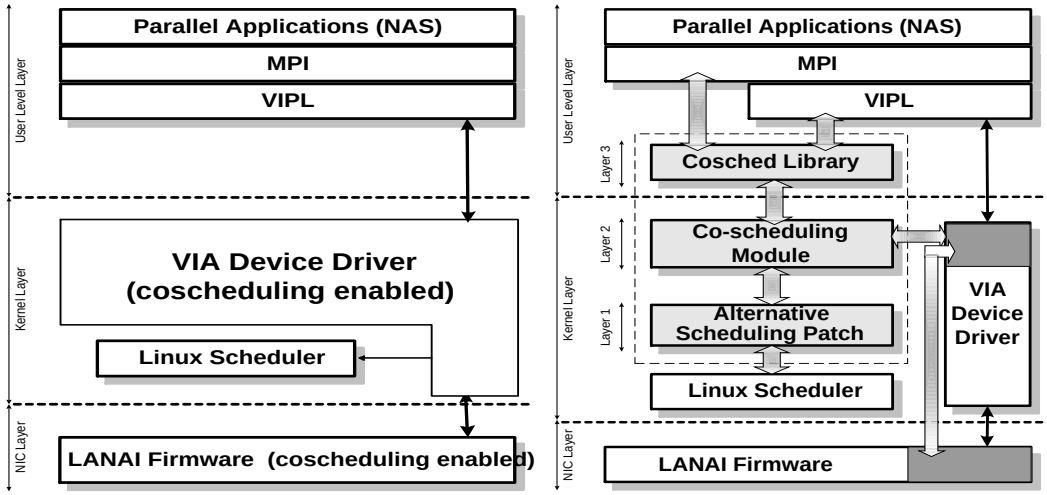
process. A slightly similar technique called Buffered coscheduling is proposed by Petrini et. al. [30]. However, it is suitable for SMP machines and not for a cluster environment, and thus, we do not discuss it further. Solsona et al. [31] implemented two demand-based coscheduling algorithms [17] on a 4 node (Ethernet connected) Linux cluster by directly instrumenting the kernel communication (*sockets*) and scheduling path. This approach is totally different from current trends, where the presence of advanced networking technology is exploited to gain much better performance. A few studies have also attempted to perform comparative analyses of several coscheduling techniques through simulation [33, 24] or analytical modeling [34]. An in depth comparative analysis through a real implementation on an 8-node solaris cluster was done in [19], where the platform is different from ours.

We feel that the lack of exhaustive experimentation (through real implementations) on several platforms using a variety of workloads is one of the primary reasons why these schemes have not yet been considered commercially viable. None of the coscheduling techniques has been tested on a cluster larger than 8 nodes (except for ICS in [15], where a 16-node implementation was done, but using a tightly-coupled programming model, as discussed above). Effect of presence of CPU or I/O intensive sequential jobs has also not been considerably evaluated in any implementation work. Only a recent simulation study [33] shows such effects but to a very limited scale, as they use only a single parallel application. Quite surprisingly, no study has also demonstrated the effectiveness of coscheduling techniques as against traditional batch scheduling technique, the primary reason why coscheduling research gained popularity in the first place.

Besides, from ours and others' experiences [16, 15, 14], we believe that implementing coscheduling techniques in a real system is non-trivial, and requires comprehensive knowledge and experience with device drivers, firmwares, OS kernel, communication libraries (VIA/U-Net/AM/FM) and parallel programming libraries like MPI, OpenMP etc. Moreover, from a few comparative studies done earlier [33, 19, 34], we infer that no one technique is so far the winner for all possible workloads and on all platforms. Hence, if a particular technique does not suit needs, or for that matter, needs change over a period of time (eg: a NIC firmware/driver is updated or a vendor or an OS is changed), complete exercise of implementing another coscheduling scheme must be done again. These factors strongly motivate us to propose a practically deployable design for coscheduling. With this motivation, we introduce our generic framework design in the next section.

3 A Framework for Implementing Coscheduling Algorithms

In this section, we introduce a generic, scalable and reusable framework for implementing coscheduling techniques. In order to appreciate the need for this framework, we first briefly examine how coscheduling algorithms have been implemented traditionally. As an example, consider the DCS [14] approach in which, when a process receives a message (intercepted by firmware), an interrupt is raised (by the firmware) for the CPU, which causes a priority boost (through the device driver) and results in immediate scheduling of that process. We observe that to implement this mechanism, support is needed from the NIC's device driver and the firmware, because only they are most accurately informed about the communication related runtime state of the parallel process(es). At a higher level, as shown in Figure 1(a), we generalize that the device driver implements the scheduling *policy* to decide which process should be executed next (typically by boosting its priority), based upon input from the NIC firmware. The NIC firmware, on the other hand, merely implements the *mechanism* to collect the information required by the device driver for this purpose. For instance, in our example of DCS, the *policy* is to boost the priority of a process for which the last message has arrived, and the *mechanism* is to generate an interrupt at



(a) Traditional Approach (b) Proposed Approach

Figure 1: Traditional and the Proposed Coscheduling Frameworks

message arrival. From this discussion, we note that a major portion of work is actually done in implementing the *policy*, and minimal work is required for implementing the corresponding *mechanism* for data collection. This insight motivates us to propose a design in which, a *policy* can be standardized and reused as a stand-alone kernel module, and proper interfaces can be outlined to implement the *mechanisms* in the firmware, whenever required. Thus, the overall idea is to cleanly abstract a coscheduling *policy* from its underlying implementation *mechanism* so that both can be treated independently. The *mechanism* designer need not know about the *policy*, as long as a clean interface is provided.

We now describe implementation of our framework in a bottom-up fashion. As shown in the Figure 1(b), our design logically comprises of 3 layers. At the lowest layer (layer 1), we implemented a generic kernel scheduler patch (Linux 2.2, Linux 2.4) that provides flexibility for the system software developers to change their local scheduling policies through an independently loadable kernel module. This patch is built upon a previous effort by Rhine at HP labs [35], but we made several enhancements including co-existence of multiple scheduling policies, application of a policy at per-process instead of per-processor level, and *mmap* system call support. Due to space limitations, we refrain from discussing the details. Next, we developed a dynamically loadable kernel module [36], called *schedAsst*, at layer 2, as shown in Figure 1(b). This module implements several re-usable coscheduling *policies*, and is one of the key contributions of this work.

Every time the local Linux scheduler is invoked by the system, before making a selection from its own *run-queue* [37], it invokes a function (__choose_task__()), implemented in the *schedAsst* module. The *schedAsst* then selects the next process to run based on certain *criteria*, and returns its recommendation to the native scheduler. The *criteria* for selection is our *policy*, and is clearly specific to the coscheduling technique enforced. Note that the implementation of this *criteria* is dependent on the necessary data provided by the firmware, through an appropriate *interface*. Details of this *interface* and an appropriate *mechanism* for its implementation in a firmware is discussed later. The native scheduler *optionally* verifies the recommendation of *schedAsst* for fairness before making a final selection decision. The final decision is then conveyed back (optionally) to the *schedAsst* for its book-keeping. Finally, we need a mechanism to let the *schedAsst* know about the specific process(es) that

would benefit from coscheduling. We provide a flexible registration mechanism through a light weight user-level library interface (*coschedLib*), implemented as our layer 3 abstraction in Figure 1 (b). The registration calls can be made directly by the application or can be transparently made by the parallel programming library (MPI) or even by the user-level communication layer. Each choice has some minor implications, but we have chosen the third option, in favor of remaining most transparent to user code.

This design clearly gives us several advantages. First, our framework can be reused to deploy and test several coscheduling mechanisms on a Linux cluster with minimal effort, which was lacking earlier. Second, the coscheduling module can be tied in easily with any user-level communication infrastructure (VIA, IBA, U-Net, GM, FM, AM etc.), simply by making some registration calls at appropriate places, a flexibility and generality not possible before. Third, this design allows us to work closely and yet independent of the native Linux scheduler, allowing us to be automatically fair across all processes in the system. Finally, by adhering to the standard interface for implementing a coscheduling *mechanism*, the driver/firmware writers can easily and independently provide coscheduling support. Moreover, by making this interface a standard across all OS platforms in future, there will be no device driver/firmware portability issues attached (with respect to coscheduling) either.

3.1 Interactions with *schedAsst* for Implementing Coscheduling Mechanisms

A typical implementation of a coscheduling *policy* requires that the scheduling entity in the kernel (*schedAsst*, in our case) is aware of the runtime characteristics of the communicating processes. Such dynamic information is most accurately captured in the NIC resident firmware, by implementing appropriate data structures and *mechanisms*. Accessing this data from a NIC in a standard way is important to allow reuse of the coscheduling module. Moreover, a well defined interface also helps firmware designers to ignore *policy* specific details and simply implement the required *mechanisms*. Since NIC resident memory is most safely accessed through its device driver, the interface is really for the device driver writer, who internally should procure any information it might need from the firmware. We have defined minimal interfaces for each of the coscheduling schemes we have implemented in our *schedAsst*, as shown in Table 3.1. The table contains three different functionalities as described below.

	Granularity	**Cosched Info**	**NIC Functionality**
DCS	VI	1. Current process ID	1. getCurrPID() 2. interruptHost(PID)
PB	VI	1. Number of pending messages	1. getNumPendingMsgs(VI) 2. incNumPendingMsgs(VI)
SB	-	-	-
CC	PID	1. Number of pending messages 2. Blocking flag 3. Message arrival timestamp	1. fetchCoschedTable() 2. updateCoschedTable()

Table 1: **Interface in the NIC to implement coscheduling mechanisms.**

Granularity directly affects the number of entries required in the table. A per-VI [3] granularity would require an entry per VI, while a per-PID granularity would require an entry per registered process. This also

[3] A Virtual Interface (VI) is a communication abstraction defined for sending and receiving messages in the VIA paradigm [8]

influences the memory used in the NIC. **Cosched Info** is the *policy* specific information, suggesting the data structure needed in the NIC to implement a particular coscheduling algorithm. **NIC Functionality** is the actual *mechanism* required to be implemented by the device driver/firmware writer to support a particular coscheduling *policy*. It should be noted that in VIA paradigm, ability to register with the NIC for an interrupt, while waiting for a message is provided as a feature, and hence SB scheme does not need explicit support from the NIC. However, it still uses the *coschedLib* and the *schedAsst* for its implementation.

Defining a clean interface like this provides added benefits. For example, coscheduling algorithm designers can easily change their heuristics by adding more specific parameters in the table (eg: Inter-message arrival rate, variance etc.) and providing appropriate interfaces to implement respective mechanisms.

4 The Proposed coscheduling approach: Co-ordinated Coscheduling

Our observation reveals that it is important to achieve co-ordination between a sender and a receiver by optimizing at both ends in order to achieve a closer approximation of perfect coscheduling. Therefore, while focusing *afresh* on how a receiver receives a message (as done by other schemes earlier [16, 15, 14]), we also carefully look at the issues involved at the sender side, and hence we call our scheme as co-ordinated coscheduling (CC).

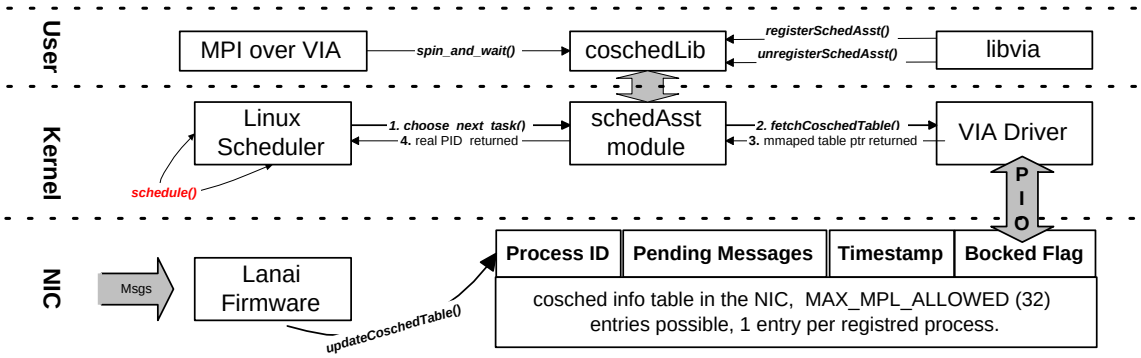


Figure 2: The Proposed Co-ordinated Coscheduling Mechanism

In the CC scheme, the sender spins for a pre-determined amount of time (*send_side_spin_time*) to wait for a *send complete* event (Not a message response, as contrary to implicit coscheduling). If a send is completed within that time, the sender remains scheduled hoping that the receiver side coscheduling algorithm will quickly have its receiver scheduled and a response can be received soon. However, if a send is not complete within this spin time, it is implicitly inferred that the outstanding message queue at the NIC is long and hence, it is better to block and let another process use the CPU. This allows for the forward progress of other parallel applications running on this node. As soon as the NIC completes the corresponding send, it wakes up the original sender process, and makes it ready to be coscheduled before the reply comes from the other end. This functionality is implemented in the *coschedLib*, and an interface (*spin_and_wait()*) is provided to be invoked by the MPI, as shown in Figure 2.

On the receiver side (See Figure 2), we maintain a per-process message arrival information table in the NIC firmware, indexed by a logical process ID ranging from 0-31. Significance of a logical PID is explained later in Section 4.1. We wait for a message arrival using spin-block again, and if a message does not arrive in

recv_side_spin_time, we block and register for an interrupt with the NIC. As shown in Figure 2, on a message arrival, the NIC firmware continuously updates this table (*updateCoschedTable()*) by recording cumulative number of incoming messages and the timestamp for the corresponding process. If the process has registered itself with the firmware, the NIC interrupts the host to wakeup this process. The woken-up process is placed in the ready queue, according to native scheduler's policies. Whenever the local scheduler gets invoked, it asks *schedAsst* module to return the next process to schedule (*1. choose_next_task()*). The *schedAsst* calls the device driver to fetch $recvd$ message information from the NIC on its behalf (2. *fetchCoschedTable()*). The device driver makes a PIO call [4] to fetch the table and makes it available to the *schedAsst* (3.). The *schedAsst* uses the table to find which process should be run next, based on whether or not a process is registered for interrupt and how many un-consumed messages are pending. After making a decision, *schedAsst* returns the corresponding process to the scheduler (4.). Note that this boosting mechanism allows us a finer control over deciding which process should get the CPU, an ability which is not possible using the original receiver-side SB [19] mechanism.

4.1 Design Choices : Analysis and Justification

Now, we discuss several critical design issues of our scheme :- *(i) How do we decide the send_side_spin_time or recv_side_spin_time ? (ii) How do we implement the exact decision policy while boosting a process ? (iii) How is the NIC firmware able to manage information at a per-process granularity ? (iv) What are the overheads associated with this scheme ?* These are summarized next.

Spin Times : The *send_side_spin_time* is dependent upon the average time it will take for the NIC to process a send, after a message enters the send queue. Usually, user-level code is unaware of this number, and hence, an intelligent approximation must be used. For example, in our Lanai firmware, we have timed that processing a single doorbell entry takes about 7-8μsec and hence, for an MPL of 4 on 16 nodes, it needs $(16 * 4 * 7)\mu secs$ to process the complete send queue in a round-robin order. This gives us a rough estimate of an average number (200 μsec). Based on what other work is pending for the Lanai, we have timed that at low loads (up to MPL 4), it takes about 150 μsec for a send to complete. Therefore, from our preliminary experimental analysis, we have found that 200 μsec is a good time within which we can get most sends completed. If send is not completed in this time, we block, assuming the presence of a bursty traffic or high workload. The *recv_side_spin_time*, on the other hand, is dependent on several factors including the network latency, the protocol processing overhead and the blocking/ context switching costs. Using the analysis done by Dusseau et al. [15], we have calculated this time to be about 300 μsec, on the platform we are using.

Boosting Decision Process : Before discussing our approach for boosting a process, let us briefly see what policies have been used earlier and what can we learn from them. In DCS, process for which a message arrives last is boosted. In our observation, at high workloads, this leads to an interrupt to the host every 100-500 μsec, leading to excessive context switching overheads and hence worse performance. In the original PB implementation [16], every 10 ms the device driver polls the head of all the RECV queues (equal to number of in-use VIs) in sequential order, and boosts (unfairly) the process corresponding to the first VI, for which a message is pending. With this approach, PB needs to poll $n * m$ descriptors (n=number of nodes, m=number of applications), giving a worst case running time of O(N^2) (sum of series : $1, 2, ...N$) , in kernel space. In SB,

[4] A Programmed-IO (PIO) call is an efficient memory mapping technique that allows fast (and atomic) word read/writes on a NIC's memory, without involving NIC processor or DMA. Please refer to [38, 37, 36] for further details.

no boosting is done at all; decision to schedule a woken-up process is left to native scheduler.

From these observations, we decided to keep a small, per-process table in the NIC (in contrast to per VI, as in DCS and PB), which accurately maintains the number of pending messages, latest timestamp and blocking information about each registered process. A constant weight ($w1$) is assigned to a blocked process(es) for which a message has already arrived, followed by a proportional weight ($w2$) to all processes based on their respective number of un-consumed (pending) messages. Adding the two weights ($w1$+$w2$) and comparing the relevant table entries provides us a fair (and a reasonably accurate) criteria to decide which process should be boosted. Note that this is only one of the several possible criteria that can be implemented using this approach, re-emphasizing its potential for further experimentation.

Per-Process Table Management : As discussed earlier, CC uses a per-process, instead of per VI table in the NIC. However, managing a per-process table in the NIC is slightly tricky, because a NIC only knows the driver assigned VI number for a communication endpoint. In order to identify the process ID to which a message belongs, we have implemented a novel zero overhead scheme, exploiting the VIA communication infrastructure. From [8], we recollect that the NIC processes the posted doorbell corresponding to each incoming message before uploading the received data (using the DMA controller). Our idea is to have user write the process ID information in the doorbell itself, and while the DMA engine uploads the data, Lanai can read the process ID from the doorbell in parallel and update the corresponding entry in our per-process table.

In order to put the correct process IDs in the posted doorbells, we exploit the lowest unused 5 bits of the 32 bit doorbell entry, reserved for future use (as mentioned in the specification [8]). Since real process IDs need more than 5 bits, we maintain a logical PID to real PID mapping internally, controlled by our *schedAsst* module. This implies that a maximum of 32 parallel processes can run simultaneously, using this approach. We believe that an MPL of 32 is sufficient for current generation of clusters at least. It is possible to explicitly write the actual PID on NIC memory, but that incurs an additional PIO [38, 37, 36] call overhead which we wanted to save [5]. In addition, it is possible to maintain other relevant information in the table to make more intelligent boosting decisions. This will allow us to devise QoS aware scheduling techniques. We refrain from discussing it further due to space constraints.

Overhead Analysis: From our discussion of the CC scheme, there are two places where we can incur overheads. First, the table fetch time should be low as it lies in the critical scheduling path. From experimental results, we find that the fetch time (including the decision making time) is not high ($2\mu sec$ for MPL4), considering the performance gains we can achieve by getting a much more accurate picture. Second, the frequency at which we fetch this table from the NIC also makes an impact. To some extent, this frequency is controlled by the frequency of the Linux scheduler invocation, but at high workloads, we find that the access frequency becomes too high (every $1ms$), and overheads start getting more than what we gain by accuracy of this information. Hence, to reduce the overhead, we fetch the table only once in $10ms$, and use a locally cached table instead, if required. Note that the fetch overhead we incur is of course quite dependent on the number of parallel processes currently running together. We timed simple fetch times up to 8 table entries (representing an MPL of 8) and it only increases to 5-6μsec. Moreover, this is our first prototype for proof of concept and we are still investigating other possibilities like cache-aware coding, optimized data encoding techniques, and dynamic adjustment of fetch frequency depending on workload, to further reduce these overheads.

[5] Note that this description assumes a 32 bit architecture. On a 64 bit architecture, there will be no such extra overhead.

5 Performance Analysis

5.1 Experimental Platform

Our experimental testbed is a 16-node Linux (2.4.7-10) cluster, connected through a 16-port Myrinet [1] switch. Each node is an Athlon 1.76 MHZ uni-processor machine, with 1 GB memory and a PCI based on-board intelligent NIC [1], having 8 MB of on-chip RAM and a 133 MHZ Lanai 9.2 RISC processor. We use the Berkeley's VIA implementation (version 0.3) [23] over Myrinet as our user-level communication layer and NERSC's MVICH (MPI-over-VIA) implementation [40] as our parallel programming library. The Berkeley VIA functionality was enhanced significantly to conform to Intel's test suite [41] and work with MVICH. We believe this is the first VIA emulator tested over a moderately large network of 16 nodes. We also tailored MVICH to put some connection establishment workarounds, necessary due to lack of stability in Berkeley VIA driver, even after our modifications. In terms of latencies, we have measured application-application round trip latency to be around 120μsec, averaged over 100,000 ping-pong messages. This includes protocol processing overheads on both the sender as well as the receiver ends. The context switching costs in Linux 2.4 kernel is about 5-6μsec [42] and interrupt processing (blocking/wakeup) costs are about 10-15μsec [43]. A programmed-IO (PIO) [38, 37, 36] call, averaged over 100 accesses, is timed to take 0.5 μsec. This is the overhead we incur when a processor reads a single word from a memory mapped location on the NIC.

5.2 Workload Characterization

Workload	Applications	Communication Intensity
$Wl1$	(EP, EP, EP, EP, EP, EP)	lo:lo:lo:lo:lo:lo
$Wl2$	(EP, EP, EP, MG, MG, MG)	lo:lo:lo:hi:hi:hi
$Wl3$	(MG, MG, MG, MG, MG, MG)	hi:hi:hi:hi:hi:hi
$Wl4$	(EP, EP, LU, LU, MG, MG)	lo:lo:me:me:hi:hi
$Wl5$	(LU, LU, LU, LU, LU, LU)	me:me:me:me:me:me

(a) Parallel Workload Composition (MPL6

Category	Workload Mix
Parallel Only	$wl1$, $wl2$, $wl3$, $wl4$, $wl5$
Parallel + CPU	($wl1$...$wl5$)+ 1 sb (1,2,4,6)sb + 2 MGs
Parallel + IO	($wl1$...$wl5$) + 1 $iobench$

(b) Executed Combinations (sb : $sched\_bench$)

Table 2: **Workload mixes used in this study.**

Our workload consists of 3 types of applications, which are typical in a cluster environment: Parallel applications, CPU intensive sequential applications and I/O intensive sequential applications. Parallel applications are further chosen to exhibit low communication, medium communication and high communication characteristics. We use $sched\_bench$, bundled with the HP's patch [35], as our CPU intensive, sequential benchmark that calculates the clock rate and megahertz of a CPU by performing floating point operations of a known duration. As our I/O intensive benchmark, we use $iozone$ [44], a file system benchmark, which intensively tests file I/O performance for several disk operations like *read, write, re-read, re-write, read backwards and read strided*. For our parallel workloads, we consider 3 applications from the NAS parallel benchmark suite [45] : EP, LU and MG; with lowest to highest communication intensities, respectively. EP is an embarrassingly parallel application that has minimal communication intensity (1%), in form of global sum computations at the master node. It exhibits a master-slave communication pattern. LU a is lower/upper triangular matrix decomposition application, which uses a large number of (16%) small (40 byte) messages. MG is a multi-grid solver that

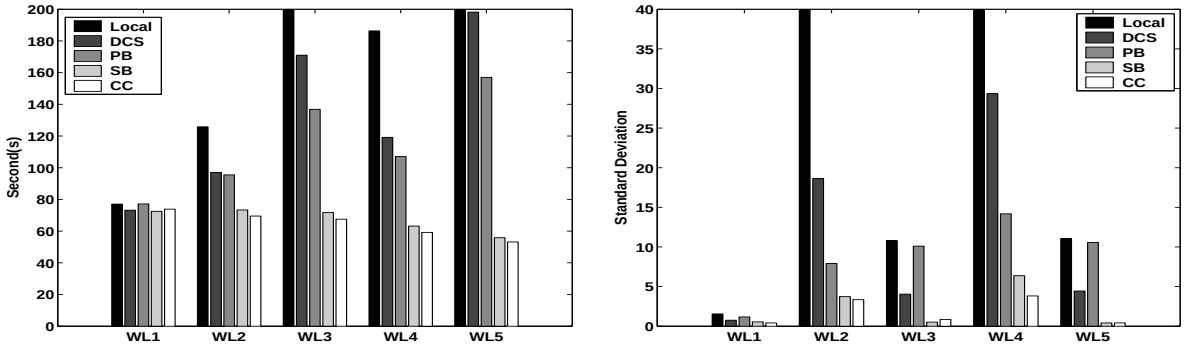
exchanges a large number of (26%) big message chunks (8K). Both LU and MG follow a ring like communication pattern. Using a combination of these 3 applications, we designed a set of 5 parallel workloads, varying in communication intensity and patterns, and is summarized in Table 2 (a).

A careful mix of these parallel and sequential workloads allows us to cover a variety of experiments up to a Multi-Programming Level (MPL) six. Note that for MPL4, $Wl4 = (EP, LU, LU, MG)$, and the ratio remains the same for other workloads, as in Table 2 (b). The complete workload space for which we have run the experiments is shown in Table 2 (b). All applications have been adjusted and compiled so they approximately take the same amount of time (10-12 sec) to complete, when executed individually. This is to ensure a fair comparison with batch scheduling, whose results we obtain by simply multiplying the individual application running time by the multi-programming degree. Total size of all applications, run together, fits well into our memory (1GB), and hence, we incur no paging overheads. Considering paging effects on coscheduling is an important area in itself, and is out of scope of this work. We analyze application execution time as the main performance metric, along with variance of execution time among competing processes, CPU utilization, and application slowdown.

5.3 Effect of Coscheduling Techniques on Parallel Applications

In this section, we analyze the effect of coscheduling techniques on the average execution time of parallel applications when run in the absence of any sequential workloads. In addition, we also measure the standard deviation of execution time among competing processes, and CPU utilization snapshots. (In order to clearly show the effectiveness of the coscheduling mechanisms, we have truncated the execution time of Local scheduling in the graphs, whenever necessary.)

5.3.1 Parallel Application Performance



(a) Average wall clock time (b) Standard Deviation

Figure 3: *Comparison of wall clock time and standard deviation of five parallel workloads (MPL=6).*

In Figure 3(a), we plot the average wall clock time of each of the five parallel workloads. It is observed that for the low communication workload $wl1$, all schemes approximately perform the same since there is hardly any need for coscheduling. As the communication intensity increases, the impact of coscheduling becomes prominent. Across all workloads, we find that SB and CC schemes perform the best, with CC getting marginally better. The reason why SB and CC show better performance is because both these mechanisms are based on a

variation of blocking technique that allows other processes to proceed, thus improving the response time as well as the throughput. The most interesting observation, however, is the fact that the SB mechanism significantly outperforms the PB scheme [6], contrary to some earlier results [16, 34, 24]. In [16, 24], PB was shown as the best performer for similar workloads and in [34], it was shown through an analytical modeling that PB performs better than SB for communication intensive workloads. In another simulation study [33], it was observed that simple, blocking techniques were more effective in some cases than spin based PB. However, our results indicate blocking-based mechanisms (SB, CC) consistently perform better across all workloads.

The main reason for this is the choice of the local scheduler. The Linux scheduler uses a simple, $counter$ controlled mechanism [37] to restrict the amount of CPU time a process receives (within a bounded time, defined by an $epoch$ [37]). This ensures fairness to all processes in the same $class$ [37]. On the other hand, priority-based, Multi-level Feedback Queue (MFQ) schedulers (Solaris, Windows NT) use a priority decay mechanism [39], where a process, after receiving a time slot, is placed in a lower priority queue and hence, can not receive CPU time unless it bubbles up at the highest level. Thus, the amount of time a process receives is not as $tightly$ controlled as it is in Linux. This means that if a process is made to remain in the highest level priority queue (which is the case in [16, 24] for PB), it is difficult to ensure fairness. In contrast, in Linux, even if a temporary boost is given (to the PB scheme) on a message arrival, this boost can not be continuously effective. This is because once the process uses its share of time slots, it can no longer be scheduled until all other processes also expire their time slots in the current *epoch*. We do not discuss the internals of Linux process scheduling and accounting mechanism for space limits, and refer interested readers to [37] for further details.

The reasons SB and CC have similar performance (although CC is marginally better) are the following. First, both of them show performance very close to the perfect batch scheduling, as we will see later. Second, the effect of the sender side blocking used in CC was marginal in the 16 node configuration. We expect that for larger clusters with higher MPL, sender side blocking would benefit more, since the number of connections exponentially increase with network size, making the firmware processing slower. Finally, even though SB and CC provide competitive performance, the proposed scheme is quite flexible and offers additional optimization opportunities. For example, we can dynamically decouple send and receive side optimizations, disable boosting mechanism or, disable the blocking mechanism, to emulate any of the coscheduling schemes like DCS, SB or even PB. This is an important flexibility towards an adaptive coscheduling mechanism design. We can even implement various boosting techniques in the NIC table for customized service through QoS guarantees.

Next, we analyze the standard deviation (SD) in the completion time of individual applications. SD is a rough measure of fairness as well as relative competitiveness of a coscheduling scheme. Primarily, a high SD signifies that the coscheduling scheme is not competitive enough, so it results in higher running times of *highly communicating* applications (like MG or LU), as compared to those with low communication (like EP). This definition especially becomes critical when we have a mixed workload like $wl2$ or $wl4$. It can also signify that a scheme is not too fair, so it completes the highly communicating jobs too early without scheduling the low communication intensive jobs. However, the fairness issue remains non-critical in a Linux environment because the native scheduler is inherently fair to all applications In Figure 3(b), we observe that the standard deviation for Local, DCS and PB schemes is rather high, especially in case of $wl2$ and $wl4$. We attribute this

[6] We varied the polling interval for PB between 1-20 ms and used 10 ms value, which gives the best performance.

effect to the degree of coscheduling. Again, we also observe that SB and CC are most competitive, and CC is doing marginally better over SB for mixed workloads ($wl2$, $wl4$).

5.3.2 CPU Utilization and Execution Time Analysis

In this section, we discuss the CPU utilization snapshots (at MPL=4) for workload $wl3$ and we further analyze $wl3$ and $wl4$ to identify various components of overall execution time. We chose only 2 workloads due to space constraints. However, these workloads should bring out the effect of homogeneous ($wl3$)and heterogeneous ($wl4$) communication intensity among various jobs. We obtain the CPU utilization information through an external program that reads the $/proc$ file system interface every 5 seconds and collects the runtime information of each application on each node. Note that results shown here are taken from a representative node, as our analysis revealed similar trend on all the 16 nodes [7].

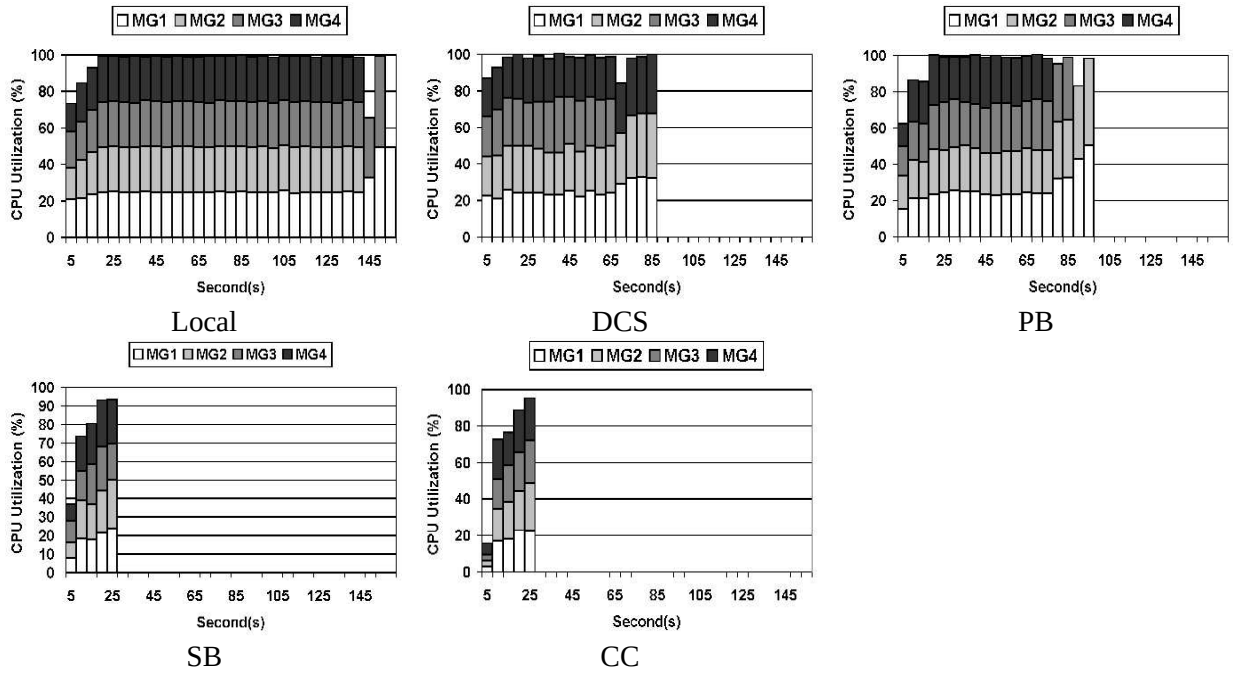
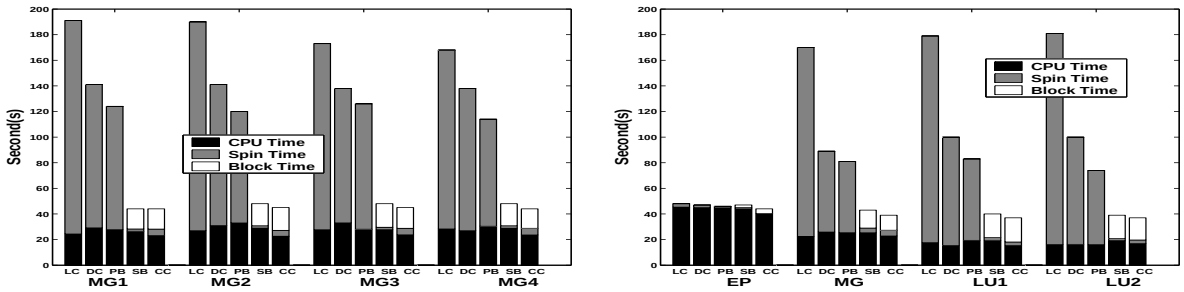


Figure 4: *CPU Utilization results for $wl3$ (MPL=4)*

As shown in Figure 4, CPU utilization across all schemes is quite evenly distributed for all applications as expected. The important observation from these graphs is that for the CC scheme, the total CPU utilization rarely peaks at 100%, (same for SB, but to a lesser extent) which suggests that there is still an opportunity to execute more parallel applications without loss in performance. Moreover, we also notice that CPU utilization during initialization phase is really low, particularly in CC and SB schemes. This is because a significant amount of time is spent waiting for a large number of connection establishments across all the nodes, and that time is better used blocking than spinning. The connection establishment time is higher than usual because we had to serialize this process in the MPI code to work around a limitation in the VIA device driver [23].

[7] Note that the numbers shown here include the overhead of running CPU and I/O intensive script(s) to collect CPU utilization snapshots and dump the results; hence can not be compared directly with earlier results.

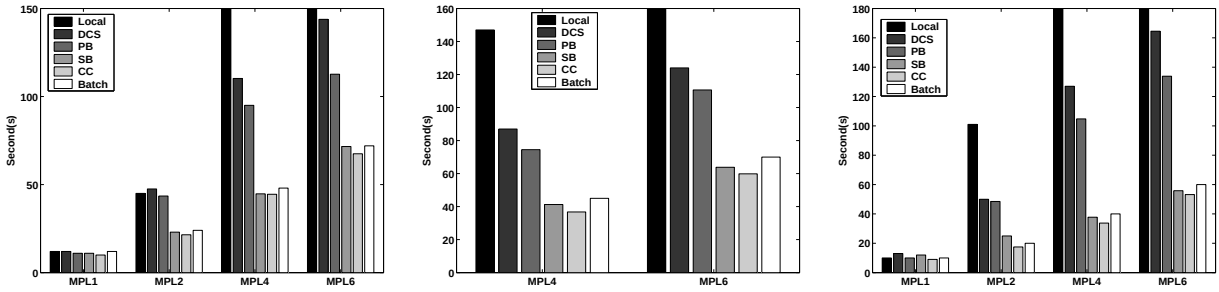


(a) Workload 3 (b) Workload 4

Figure 5: *Detailed Execution Time analysis for $wl3$ and $wl4$ (MPL=4).*

An important aspect often not considered while viewing CPU utilization graphs is, how much of that time is actually being used for useful computation and how much is being wasted in spinning. In Figure 5, we plot a breakup of useful work done (actual useful CPU time), spin time and block time (time spent waiting off-line) of each application in $wl3$ and $wl4$. The sum of all these times gives the wall-clock time, but it includes some additional overheads (not shown) for the data collection mechanism. (Note that the block time shown here is *not* an overhead on the CPU.) In both the workloads, we observe that for Local, DCS and PB, most of the CPU time is wasted due to spinning, and this has been effectively reduced by both, SB and CC schemes. These schemes block instead, allowing other applications to use the CPU. As expected, the actual useful work done for all schemes is approximately the same; the marginal differences are attributed to other overheads.

5.4 Effect of Multi Programming Level



(a) Workload 3 (b) Workload 4 (c)Workload 5

Figure 6: *Effect of MPL on the average wall clock time of various workloads.*

With increase in processor speeds, memory density, improvements in the network bandwidths, and most importantly, much reduced context switching/interruption overheads due to highly sophisticated and optimized OS kernels (like Linux), we have potential for fully utilizing the CPU through a high degree of multi-programming. In Figure 6, we observe that as we increase the MPL, Local, DCS and PB schemes have a very steep increase in execution times, whereas SB and CC are much more tolerant. The effect is more prominent for workloads $wl3$ and $wl5$, (other workloads are not included here) suggesting that for medium to high communication oriented workloads, SB and CC can adapt and coschedule relatively easily. This is because as we increase the MPL, the likelihood of processes remaining coscheduled for a longer time gets lesser. This makes blocking and wakeup

on demand a better option than spinning.

Another significant observation from these graphs is that as we increase the MPL, we can start seeing the benefits of a good coscheduling technique against batch scheduling. SB and CC are doing at least equal to or sometimes better than batch scheduling at MPL2, and the gains getting more prominent at MPL4 and MPL6. Since this is a very important conclusion of this work, we discuss this further in the next subsection.

5.5 Comparing Batch Scheduling to Coscheduling

	MPL-4						MPL-6					
	Local	**DCS**	**PB**	**SB**	**CC**	**Batch**	**Local**	**DCS**	**PB**	**SB**	**CC**	**Batch**
$wl1$	53	54	51	53	50	52	79	74	78	73	74	78
$wl2$	103	75	71	47	47	50	175	115	104	72	72	75
$wl3$	191	111	124	45	44	48	300	174	145	68	68	72
$wl4$	181	100	83	47	42	45	253	144	121	71	63	70
$wl5$	276	128	108	38	35	40	452	202	166	56	54	60

Table 3: **Last Application Completion time (seconds) of 5 parallel workloads with different schemes**

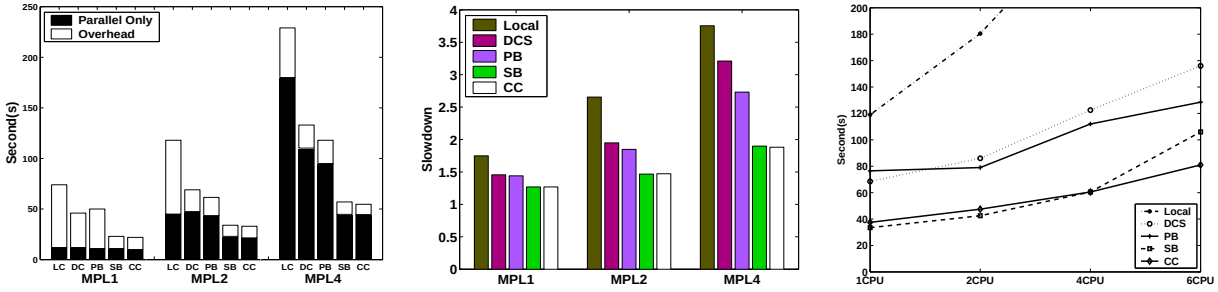
In Table 3, we show the worst case completion time (time when last application completed) of all workloads using various schemes (including batch processing). Note that for batch scheduling, we simply add up the running time of all the applications in a workload. We see that CC is the most competitive scheme which can provide performance very close to batch scheduling, with SB close behind. The reason is that both these schemes efficiently exploit idle time opportunities by blocking whenever necessary, and thus benefit from low interrupt processing (10-15 μsec) [43] and context switching overheads (5-6 μsec) [42] and a fair Linux scheduler. Moreover, this result assumes a zero overhead global batch scheduler and a perfect allocation without node fragmentation; hence it is a comparison to the best case batch scheduling. This is the first encouraging result directly in favor of coscheduling as compared to batch processing. To the best of our knowledge, this effect has not been demonstrated in coscheduling research, specifically for a cluster environment.

5.6 Effect of Sequential, CPU and I/O Intensive Applications

We next consider workloads consisting of both, sequential (CPU, I/O) and parallel applications, representing a typical scenario on a time-sharing cluster. This effect was only observed earlier in [33] on a smaller scale as they simulate using only a single parallel application. We measure the average execution time, overhead and the slowdown of a job due to the presence of another. We define slowdown of a CPU(I/O) intensive job as the ratio of the running time of the job in the presence of parallel applications to that in the absence [8]. We also define overhead as the difference between the running time of parallel jobs in the presence of CPU(I/O) intensive jobs and the corresponding running time in their absence [9]. We observe that for low communication intensive workloads ($wl1$), presence of CPU or I/O jobs slows down all processes linearly, as we increase the MPL. Since the result is quite expected, we do not show them to save space. We only discuss their effects in the presence of a communication intensive workload ($wl3$), shown in Figures 7 and 8.

[8] slowdown of CPU or I/O job = (running time CPU(I/O) + parallel)/ (running time CPU(I/O) only)

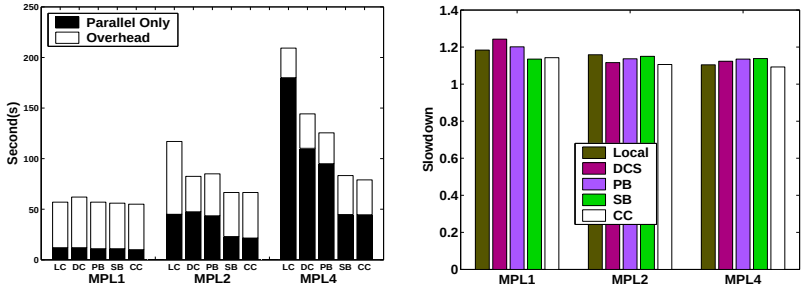
[9] overhead of Parallel Job = (running time parallel + CPU(I/O)) - (running time parallel only)



(a) Effect on parallel job (b) Effect on CPU job (c) Scalability with CPU jobs

Figure 7: *Performance Analysis of Parallel ($wl3$) and CPU Intensive application mix.*

From Figure 7(a), we see that execution time of parallel jobs increase in the presence of a CPU intensive job, as expected. However, if we examine the overhead part across all schemes, we notice that SB and CC incur significantly less overhead than others. The results indicate that blocking based coscheduling schemes can tolerate presence of CPU bound jobs and still remain coscheduled nicely. Further, as we increase the number of CPU jobs (as shown in the Figure 7(c)), we see that CC shows better tolerance than others. This is because of the effectiveness of the boosting mechanism in the CC policy, which is not present in the SB. Finally, in Figure 7(b), we observe that the rate of increase in slowdown is smaller for SB and CC than others. The asynchronous nature of the algorithms like CC and SB allows them to efficiently exploit idle time and hence they have a minimal impact on the CPU job as compared to other schemes.



(a) Effect on parallel job (b) Effect on I/O job

Figure 8: *Performance Analysis of Parallel ($wl3$) and I/O Intensive application mix.*

Next, we experiment with a mix of parallel workloads and sequential I/O jobs, and plot the results in Figure 8. An interesting observation is that I/O intensive jobs remain unaffected in the presence of parallel jobs. This is true irrespective of the coscheduling scheme used. This is primarily because inherent high priority given by the scheduler to an I/O intensive process does not provide enough opportunity for coscheduling. Accordingly, from Figure 8(a), we can see that parallel jobs get more affected in the presence of I/O intensive workloads than due to CPU workloads. We could not bring in effect of varying number of I/O jobs in presence communication intensive workload like $wl3$. This is due to the scalability limitations of the prototype Berkeley VIA device driver. However, we expect that as number of I/O jobs increases, performance of communication intensive parallel applications will suffer further. This is also in line with the observation made in [33].

6 Conclusions

Communication-driven coscheduling, although has been shown to improve the performance of parallel applications in a multi-user cluster environment, has not made its way into commercial systems. Most large clusters still use variations of batch processing to schedule parallel jobs. The primary motivation of this paper is, therefore, to address some important issues for developing practical scheduling alternatives for clusters. Specifically, we focus on Linux clusters that dominate the current market place.

As a first step in this direction, we have proposed a generic framework that can be used to implement any coscheduling algorithm with minimal effort. We have demonstrated the flexibility and modularity of this framework by implementing three prior coscheduling techniques (DCS, SB and PB) and a new technique, called Co-ordinated coscheduling (CC). The new scheme, unlike the prior policies, uses a combination of blocking at both the sender and receiver ends and a boosting mechanism. The boosting mechanism is intelligently implemented with the help of the Myrinet NIC firmware, and optimized for low-overhead by utilizing the unused bits of the VIA doorbell structure. A complete prototype has been implemented on a 16-node Myrinet connected Linux cluster using the VIA communication paradigm. Next, exhaustive experiments using a variety of parallel and sequential workloads have been conducted to analyze the relative merits of the four coscheduling algorithms, and have been compared against the base case local Linux scheduler and the ideal batch scheduler.

The important conclusions of this paper are the following: (i) In contrary to the prior research [16, 19, 24, 34], we observe that SB and the proposed CC algorithms outperform the PB scheme over all workloads in a Linux cluster. This is primarily attributed to the fair scheduling policy of the native Linux OS and the asynchronous nature of the two schemes; (ii) The SB and CC schemes can compete with and perform even better than the batch processing at a reasonable multiprogramming level of four to six. This suggests that these techniques are viable candidates for further consideration; (iii) The proposed CC algorithm shows equal or better performance compared to the SB scheme on the 16-node platform. Moreover, it is quite general in the sense that one can optimize both the sender and the receiver sides independently for boosting performance. The sender side blocking will be more effective as the network scales. In addition, the clever table implementation in the NIC has the potential to incorporate a variety of boosting techniques including that for QoS; (iv) The proposed framework addresses the ease of implementation and portability issues in that one needs to change only the driver interface to make it work. Although, we could not address the scalability issue due to lack of access to a larger cluster, we believe that the proposed framework has no scalability problem as long as the underlying communication mechanism scales well. In this context, we are not sure of the scalability of Berkeley VIA and may use other available commercial implementations such as Myrinet's GM [1] or Emulex's VI-IP over $cLan$ [2] etc for future.

This prototype design has been a major exercise that required significant work at several levels of design (MPI library, VIPL, VIA driver, VIA firmware, Linux scheduler and a kernel based coscheduling module). However, we believe that it will provide a strong foundation for investigating and optimizing various coscheduling design alternatives. As an interesting exercise, we would like to show how the CC algorithm can adapt to the native scheduler to maximize performance. It was shown in [16, 34] that PB is a better choice for a multi-level priority queue based scheduler (Solaris, Windows NT), while in this paper we showed that a scheme based on blocking (SB and CC) can provide the best performance in a scheduler like that of Linux. The proposed CC can be easily used in both the environments. We have observed that by disabling the blocking scheme and simply

using the boosting, CC is closer to PB and shows slightly better performance on the Linux platform. On the other hand, by enabling the blocking mechanism, it has the ability to provide better performance than SB.

Our research has brought out several interesting issues for further investigation. For example, implementing this module on several platforms and examining the scalability on a larger platform will be important for real deployments. In this context, we plan to port this module to GM and examine the scalability on a larger platform. This will be done in collaboration with the Lawrence Livermore National Laboratory (LLNL) and IBM to test on their large clusters. Second, the proposed CC scheme's ability to scale with higher MPL needs further investigation. We are looking at intelligent cache-aware coding and data compression techniques to optimize the access of critical data on the NIC. Moreover, exploration of several new heuristics to perform intelligent coscheduling decisions using our *generic table* approach is a new direction with immense potential for QoS support. Once stable and tested for scalability, we plan to make this work available in the public domain.

References

[1] N. J. Boden *et al.*, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29–36, February 1995.

[2] Emulex Corp., "The VI/IP Standard and cLan." Available from `http://www.emulex.com`.

[3] G. E. Alliance, "10 gigabit ethernet technology overview white paper." Available from `http://www.10gea.org/Tech-whitepapers.htm`.

[4] Dolphin Networks., "Dolphin SCI Technology." Available from `http://www.dolphinics.com`.

[5] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proc. 15th ACM Symp. on Operating System Principles*, pp. 40–53, Dec. 1995.

[6] S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," in *Proceedings of Supercomputing '95*, December 1995.

[7] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, May 1992.

[8] Compaq, Intel and Microsoft Corporations, "Virtual Interface Architecture Specification. Version 1.0," Dec 1997. Available from `http://www.vidf.org`.

[9] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, vol. 15, pp. 54–64, February 1995.

[10] J. K. Ousterhout, "Scheduling Technique for Concurrent Systems," in *Int'l Conf. on Distributed Computing Systems*, pp. 22–30, 1982.

[11] D. G. Feitelson and L. Rudolph, "Gang Scheduling Performance Benefits for Fine-Grained Synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, December 1992.

[12] D. G. Feitelson and L. Rudolph, "Coscheduling Based on Runtime Identification of Activity Working Sets," *International Journal of Parallel Programming*, vol. 23, pp. 136–160, April 1995.

[13] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette, "Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific," in *Proceedings of Supercomputing*, November 1999.

[14] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic Coscheduling on Workstation Clusters," in *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 231–256, March 1998. LNCS 1459.

[15] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "Scheduling with Implicit Information in Distributed Systems," in *Proc. ACM SIGMETRICS 1998 Conf. on Measurement and Modeling of Computer Systems*, pp. 233–243, 1998.

[16] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. Das, "A Closer Look at Scheduling Approaches for a Network of Workstations," in *Proc. of the ACM Symposium on Parallel Algorithms and Architectures(SPAA)*, pp. 96–105, June 1999.

[17] P. G. Sobalvarro and W. E. Weihl, "Demand-based Coscheduling of Parallel Jobs on Multipr ogrammed Multiprocessors," in *Proc. IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 63–75, Apr. 1995.

[18] M. Buchanan and A. Chien, "Coordinated Thread Scheduling of Workstation Clusters under Windows NT," in *Proc. of USENIX Windows NT Workshop*, Aug. 1997.

[19] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das, "Alternatives to Coscheduling a Network of Workstations," *Journal of Parallel and Distributed Computing*, vol. 59, pp. 302–327, November 1999.

[20] "LoadLeveler." `http://www.mhpcc.edu/training/workshop/loadleveler/`.

[21] PBS Pro, "Portable Batch System (PBS)." Available from `http://www.pbspro.com`.

[22] InfiniBand Trade Association, "InfiniBand Architecture Specification, Volume 1, Release 1.0," October 2000. Available from `http://www.infinibandta.org`.

[23] P. Buonadonna, A. Geweke, and D. E. Culler, "An Implementation and Analysis of the Virtual Interface Architecture," in *Proceedings of Supercomputing '98*, November 1998.

[24] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke, "A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment," in *Proceedings of the ACM 1996 International Conference on Supercomputing*, pp. 100–109, May 2000.

[25] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 401–420, April 1994.

[26] S. T. Leutenegger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," in *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pp. 226–236, 1990.

[27] E. W. Parsons and K. C. Sevcik, "Coordinated Allocation of Memory and Processors in Multiprocessors," in *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pp. 57–67, June 1996.

[28] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors," in *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pp. 214–225, 1990.

[29] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective Distributed Scheduling of Parallel Workloads," in *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pp. 25–36, 1996.

[30] F. Petrini and W. C. Feng, "Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems," in *In Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 439–444, 2000.

[31] F. Solsona, F. Gin, P. Hernndez, and E. Luque, "CMC: A Coscheduling Model for non-Dedicated Cluster Computing," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[32] National Energy Research Scientific Computing Center, "MVICH - MPI for Virtual Interface Architecture," 2001. Available from `http://www.nersc.gov/research/FTG/mvich/`.

[33] C. Anglano, "A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations," in *Proceedings of 9th International Symposium on High Performance Distributed Computing (HPDC'9)*, August 2000.

[34] M. Squillante, Y. Zhang, A. Sivasubramaniam, H. Franke, and J. Moreira, "Modeling and Analysis of Dynamic Scheduling for Parallel and Distributed Environments," in *Proceedings of the ACM SIGMETRICS 2002 Conference on Measurement and Modeling of Computer Systems*, pp. 43–54, June 2002.

[35] H. P. Scott Rhine, MSL, "Lodable scheduler modules on linux white paper." Available from `http://resourcemanagement.unixsolutions.hp.com`.

[36] A. Rubini and J. Corbet, "Linux Device Drivers, 2nd Edition." Available from `http://www.oreilly.com/catalog/linuxdrive2/`.

[37] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel." Available from `http://www.oreilly.com/catalog/linuxkernel/`.

[38] Caldera Inc., "Uniform Driver Interface (UDI) Specifications 1.01." Available from `http://docsrv.caldera.com/UDI_spec/physical_io_spec-2.html`.

[39] SUN Microsystems Inc., "Solaris 2.6 Software Developer Collection Vol 1: System Interface Guide, 1997." Available from `http://www.sun.com/`.

[40] National Energy Research Scientific Computing Center, "M-VIA: A High Performance Modular VIA for Linux," 2001. Available from `http://www.nersc.gov/research/FTG/via/`.

[41] Intel Corp., "The Intel Virtual Interface Architecture - Developers Guide." Available from `http://developer.intel.com/design/servers/vi/developer/ia_imp_guide.htm`.

[42] Dr. Edward G. Bradford, IBM, "Measuring the Scheduler Overhead." Available from `http://www-106.ibm.com/developerworks/linux/library/l-rt9/`.

[43] A. B. Maccabe, “Interrupt Latency Timer using Myrinet.” Available from http://www.cs.unm.edu/ maccabe/SSL/.

[44] O. C. Willaim D. Norcott, “Iozone fi le system benchmark white paper.” Available from http://www.iozone.org.

[45] N. A. S. division., “The NAS parallel benchmarks (tech report and source code).” Available from http://http://www.nas.nasa.gov/Software/NPB/.